A Protocol for Wait-Free, Atomic, Multi-Reader Shared Variables

Richard Newman-Wolfe
Computer and Information Sciences Department
The University of Florida

Abstract

A wait-free solution to the atomic *(r,1)*-CRWW problem is presented. It uses *(r+2)(3r+2+b)-1* safe, multi-reader bits to implement a *b*-bit variable shared among *r* readers and *1* writer, solving an open question of Lamport [Lamport '85]. Thus, it closes a gap in the simulation of systems of shared variables for single writers. It differs from other approaches to this problem in that mutual exclusion is maintained between the writer and other processes on the copies of the shared variable used in the implementation.

Introduction

Sharing resources among asynchronous processes is a fundamental problem in parallel and distributed systems. By their nature, many kinds of resources require exclusive access, which leads to the mutual exclusion problem [Dijkstra '65]. Mutual exclusion ordinarily involves waiting by one process if another is currently using the resource. Since a major goal of parallel systems is to decrease the time required to complete a set of tasks by doing them concurrently, it is important to decrease or eliminate waiting whenever possible.

The requirements of shared data are different than those for other types of resource. For instance, a process only wishing to read the data (a *reader*) should not prevent another such process from accessing the data simultaneously. This was recognized and presented as the *readers/writers* problem [Courtois, Heymans & Parnas '71]. A solution to this problem requires only that any number of readers be allowed access to a single copy of the shared data. Mutual exclusion between readers and *writers* (processes wishing to modify the data) and between writers and writers must be maintained, so there would never be the problem of a read obtaining an incorrect value from the actions of one or more overlapping writes.

A second version of this problem that only dealt with a single writer was introduced by [Lamport '77]. The *concurrent reading and writing (CRAW)* problem permits simultaneous access by the readers and the writer to a single copy of the shared data, requiring only that a reader be able to tell if an overlapping write occurred

during its read. In the solution given, while the writer never waits, the readers may be locked out by a fast writer, since the reader must discard the potentially corrupted value it read and try again.

The *concurrent reading while writing (CRWW)* problem requires not only that readers and writers be able to access the shared data concurrently, but that the reader be able to return a "recent" and accurate value, regardless of whether or not an overlapping write occurred [Peterson '83a], [Lamport '85]. While the solution using mutual exclusion satisfies this requirement, the case of greatest interest is that in which no process waits on any other process. In order to achieve this, multiple copies of the shared data must be used. Clearly, it is desirable that there be no waiting when there are no assumptions about the relative speeds of the processes.

In the solution presented here, the shared variable is simulated by the protocols given, using two or more buffers to hold copies of the shared variable. Access to the buffers is controlled by concurrently accessed control variables. It is important to note that the variables comprising the buffers and the control variables are all strictly weaker than the type of variable being simulated. It is the ability to "strongly simulate" a system that uses the stronger variables by a system comprised of weaker ones that makes this work of theoretical interest.

Preliminaries

For many problems, the shared resource can be modeled as a variable accessed by several processes of two types: readers and writers. Only writers may change the value of the variable, which can be read simultaneously by many readers. A *CRWW protocol* is an algorithm for simulating a shared variable accessed by readers and writers, all of which may be accessing the variable simultaneously. The problem lies in guaranteeing certain "fairness" and "safeness" properties for the variable the protocol implements. In general, if there are w writers and r readers and the variable implemented is "atomic", this is referred to as the *(r, w)-CRWW* problem.

There are several dimensions of "strength" for systems of shared variables. Strength refers to the nature of the assumptions a user of the system may make about the behavior of the variables used: the users of a "strong" system may make more assumptions about the variables in their system than the users of a "weak" system. The dimensions of these assumptions include the number of writers permitted per variable, the number of readers per variable, the number of values per variable, the fairness of access to the variable, the safeness properties of the variable, and whether the system is synchronous or not. This paper extends the results by Lamport and Peterson in simulating powerful variables by using weaker ones.

Lamport has identified several categories of shared variable in terms of the interaction between one writer and one or more readers [Lamport '85]. These *safeness properties* are, in order of increasing strength, *safe, regular* and *atomic*. The only assumption we may make about a *safe* variable is that a read that does not overlap a write will return the most recent value written to the variable. If a write overlaps the read, the value read may be meaningless. Informally, let us say that a value is *valid* in interval I if it is either the last value that was written to the variable prior to I, or if it is a value being written

to the variable by a write that overlaps *I*. Then a *regular* variable has the property that a read will return some value that is valid in the interval over which the read takes place. In particular, this definition allows one read to return an "new" value and a strictly later read to return the "old" value. Regular variables have also been called "flickering", to evoke the image of the variable's value flickering between the old value and the new value over the duration of the write [Peterson '83b]. (This is a reasonable model of mechanical switches having contact bounce, for instance.) *Atomic* variables behave as though operations on them occur at some instant within the interval over which the operation takes place. Since we are concerned only with single-writer variables here (*distributed variables*, as introduced in [Lamport '74]), we need not consider multi-writer safeness properties [Newman-Wolfe '86b].

Fairness properties have to do with the amount of waiting a process may have to endure in order to complete its access. At the extreme of unfairness, there are the well-known conditions of deadlock, livelock and starvation, all of which imply that one or more processes may never gain access to the shared resource. Bounded waiting sets some upper limit on the number of times other processes may access the shared resource after a given process has indicated its desire to access the resource. At the other extreme, a protocol may be *wait-free*, meaning that a process will gain access to the shared resource regardless of the actions or speed of any other process. Other processes cannot prevent a wait-free process from successfully completing its access. Since there are different types of process in the CRWW problem, it is possible for one type of process to have one fairness property and another type to have a different one. A CRWW protocol is *writer-priority* if the writer never has to wait on a reader [Lamport '77].

The two most fundamental characteristics of the solution presented here are that it is wait-free and that the variable implemented is atomic.

Previous Results

Early solutions [Courtois, Heymans & Parnas] of the CRWW problem simply used mutual exclusion, enforced by semaphores. This is overly restrictive because of the unnecessary waiting it introduces. Furthermore, the implementation of semaphores is itself a problem of atomic shared variables, and so begs the question.

Lamport introduced the first writer-priority, atomic *(r,1)*-CRWW solution that used regular shared variables [Lamport '77]. His solution used only one buffer but had control variables that had to hold arbitrarily large values; it was also possible for the readers to starve.

A wait-free solution was given by [Peterson '83a]. However, this required two atomic control bits. Peterson's wait-free solution uses *r+2* copies of the shared variable, which he shows to be required for any wait-free solution. His construction produces an atomic buffer for one writer and *r* readers using *2r* atomic single-reader bits; two atomic, *r*-reader bits; and *b(r+2)* safe *r*-reader bits, where *b* is the buffer size (the number of bits in the variable). One difficulty here is that it was not known how to make wait-free, atomic, *r*-reader bits from weaker variables. Peterson also gave solutions that used fewer buffers but allowed various types of waiting.

My earlier work included an algorithm that gave a tradeoff between the maximum number of readers on which a write could wait and the number of buffers used [Newman-Wolfe '86a, '86b]. The algorithm implements a writer-priority solution if $r+2$ buffers are used, with the writer's general space-waiting tradeoff being

$$(space-1)\times(waiting)=r,$$

where space is measured in the number of buffers required, and waiting is the upper bound on the number of readers for which the writer may have to wait during a single write. This algorithm used only safe, single-writer, multi-reader bits to implement an atomic, single-writer, multi-reader multi-valued variable. With appropriate extensions of the safeness properties to writer-writer interactions, the results were extended to multi-writer variables. A deficiency was that in all cases, it was possible for readers to wait on a writer.

Recently, Lamport constructed the first atomic, wait-free variable using only regular variables in its implementation [Lamport '85]. The variable built was for a single reader and a single writer. In the same paper, he provided a construction for a regular, wait-free, single-writer, multi-reader, multi-valued variable from regular bits. This construction is used for part of the solution given here.

The multi-writer, multi-reader, atomic solution reported in [Vitanyi & Awerbuch '86] has since been withdrawn by Awerbuch. In any case, it required both readers and writers to make $r + w$ copies of the shared variable and an accompanying timestamp. Their version using arbitrarily large time stamps appears to be correct, using n^2 n-reader, single-writer, regular variables, where n is $r + w$. If this is translated to safe bits, the total amount of space used is quite large, even if some "lifetime of the universe" argument is used to put a bound on the size of the timestamps.

At around the same time as the protocol presented here was discovered, Burns and Peterson developed a single-writer, multi-reader, wait-free, atomic algorithm [Burns & Peterson '87]. They use two single-writer, multi-reader, multivalued, regular copies to implement the desired variable. The most space-efficient version uses this new protocol to implement the atomic bit required by the wait-free algorithm of [Peterson '83a].

When reduced to single-writer, multi-reader, safe bits, their protocol does use less space (both absolutely and asymptotically) than the one presented here, but there are two differences worth noting. One is that the solution here preserves mutual exclusion between the writer and other processes on the copies of the shared variable. This is interesting from a philosophical point of view. Second, from a more practical standpoint, the earlier Peterson CRWW algorithm requires the writer to make a copy for each reader that started it read since the *previous* write started; the writer may have to make many copies for readers that are no longer trying to access the variable, and are hence unnecessary. There appears to be no simple remedy for this deficiency. The protocol presented here always makes at least two copies of the shared variable, but never does it make any additional copy unless it actually encounters an active reader during its write.

The algorithm presented in this paper requires only safe, single-writer, multi-reader bits to implement a wait-free, atomic, single-writer, multi-reader, multi-valued variable. The writer may have to write up to $r+1$ copies of the shared vari-

able and a few control bits per reader, but no reader has to read more than one copy of the shared variable or write more than two control bits per read. This is slightly less work than that potentially required by the wait-free solution given in [Peterson '83a], where the writer may have to write up to $r+2$ copies of the shared buffer per write and the reader always reads at least two and may read as many as three copies of the shared variable. It is much more work than is required by the writer-priority solution of [Newman-Wolfe '86a], where the writer writes exactly one copy and the reader reads exactly one copy.

The Main Result

From an historical point of view, the result in this paper followed from combining ideas in [Peterson '83a], [Lamport '85] and [Newman-Wolfe '86a]. A brief overview of the key ideas may be useful to the understanding of the current result.

Peterson's construction utilized a primary and a secondary buffer shared by all readers, and a private buffer for each reader, for a total of $r+2$ copies. The writer wrote the primary, then made a private copy for each reader that started since the last write, then wrote the secondary. The readers first read the primary, then the secondary, then determined from the control bits they read which of these to use or whether to use the private copy. The copy used was guaranteed to have been read without an overlapping write, although one or both of the others could have suffered on overlapping write.

Besides the two bits used to allow the reader to determine whether a write overlapped its read of the primary buffer, there are a pair of bits per reader used to signal the reader if a private copy of the variable has been made for it. Following the terminology of [Peterson & Burns '87], these will be called *forwarding bits*. One of the bits is written by the writer, who tries to make the two bits equal (*off* or *clear*), while the other bit is written by the reader, who tries to make them not equal (*on* or *set*). This simulates a two-writer bit using two distributed bits. In Peterson's algorithm, this pair of bits is cleared by the writer whenever the writer makes a private copy for a the reader, and is set by the reader whenever it starts a read. Thus, if the bits are equal by the time a reader reaches the end of its read, it knows that the writer has made private copy of the variable for it since it started its read. A similar technique will be used to permit communication between readers in the algorithm given here.

The algorithm of [Newman-Wolfe '86a] also works by using multiple copies of the simulated variable. All buffers are identical, however, unlike the multiple buffers used by Peterson. The copy holding the current value is indexed by a regular register written by the writer, called the *selector*. The protocols used insure that no reader is reading a buffer while the writer is changing it. This is similar in technique to the use of "shadow copies" for transactions, but much more care has to be taken due to the absence of atomic control variables (in particular, the selector is not atomic).

That implementation uses an $(M-1)$-bit regular register written only by the writer, that selects the current copy. The selector register is implemented by Lamport's wait-free, multi-reader, regular register from safe bits [Lamport '85]. For each copy there is a control bit written by the writer and r control bits written by the

readers (one per reader per copy). If each copy has b bits, the total number of safe bits used for the algorithm is $M(2+r+b)-1$. With enough buffers, the writer never has to wait, but the readers may have to wait no matter how many copies are used. The object of the construction given here is to eliminate any possibility for the readers to wait.

This end is achieved by adding a backup copy associated with each primary copy, so the buffers now come in pairs (refer to figure 1 for a high-level view of the algorithm). For each pair of buffers, there are a number of control bits. In addition to a read flag for each reader and a write flag for the writer, there are a pair of forwarding bits per reader that are used in a set and cleared as explained above. Lamport conjectured that it was necessary for the readers to communicate with each other in order to implement a multi-reader atomic variable [Lamport '85]. The construction given here does this through the use of these forwarding bit pairs. Their utility lies in informing each reader if an earlier reader has determined that the writer is through with a pair of buffers and that all subsequent readers may freely access a newly written copy.

There are two keys to using these tools correctly. First, any reader that accesses a new value must somehow inform later readers that they too may access the new value. Second, the way in which the backup copies must be used is critical. It will not do to write the new value to the backup copy, since the same problems exist with it as existed with the single copy version. Instead, the most recent previous value is written to the backup copy, so that readers accessing the selector variable while the writer is changing it will get the same value regardless of what they read. Those reading the new value for the selector will be forced to read the backup copy, while those reading the old value will be able to access the primary copy of the old pair of copies. These two buffers hold identical values, so there is no danger of reading a new value, then a strictly later read reading the old value.

By the time a reader obtaining the new value for the selector is allowed to read the primary copy of the current pair, the value of the selector is stable, so all subsequent readers will obtain the new selector value. The only problem is to insure that once a reader has accessed the primary copy of the new buffer pair, all subsequent readers will also access the primary copy of the new buffer pair. This is the purpose of the forwarding bits. Before the writer writes the new value to the primary buffer, it clears all the forwarding bit pairs. The first (and any) reader that accesses the primary buffer will set its forwarding bit pair. Readers that see the write flag on check the forwarding bits for all readers; if any have been set, the reader infers that some previous reader saw the write flag off so it is not only all right but imperative to read the primary copy.

The writer's protocol may be thought of as consisting of three phases. These phases are separated by a check of all of the read bits for the pair of buffers the writer is using. During each successive phase, the writer is assured of additional facts about the state of the readers that may be interested in the same pair of buffers.

First, the writer finds a pair of copies apparently free of readers. At this point, the writer knows that no straggling readers could have seen its write flag on for this

pair of buffers. The writer writes the most recent previous value to the backup buffer, then sets its write flag. Again, the writer checks for straggling readers. If any are found (via read flags), then it goes back to the beginning, after clearing its write flag. Otherwise, it enters phase 2, in which it is assured that any readers that turn on their read flag *must* see its write flag on. The writer clears that buffer pair's forwarding bits for each reader. It then checks for stragglers one final time, including a check of the forwarding bits. If any are set, it goes back to the start, clearing its write flag. When it passes the third check, it enters phase three, where it knows that any reader that could have seen the forwarding bits on must have finished, and that any reader setting its read flag must not only see the write bit on, but must also see the forwarding bits clear. The writer then writes the primary buffer, changes the pointer, and clears its write flag, remembering this value so it can use it to write the backup copy the next time it does a write.

The proof that the algorithm is correct hinges on two key observations. First, the writer never modifies a buffer while a reader is accessing it (Lemmas 1 and 2). Second, the coordinated use of write flag and forwarding bits prevents any strictly sequential reads from obtaining a new value then an old value (Lemma 3). These provide the basis for the final statement of the result in Theorem 4. The only additional requirement to be shown for Theorem 4 is that the algorithm is wait-free.

The only readers of any consequence to the writer during phases 1 and 2 are those readers that read the selector before the current write began, when the selector was pointing to the buffer pair in which the writer is currently interested. These we will call *old* readers. *New* readers are those readers that read the selector after the writer starts to change it but before the writer finishes clearing its write flag. They are the only other readers of consequence to the writer, since readers that read the selector after the writer begins and before the writer changes the selector will only access the current buffer pair, and will in no way interfere with the writer or attempt to access either of the buffers the writer changes.

For the purpose of the proofs below, these readers will also be categorized according to the time at which they turn their read flag on. Those that complete turning their read flag on before the writer's first check must also finish their entire read before the writer performs the first check, or else the writer will not attempt to access that pair of buffers at all. These readers will not attempt to access either buffer at the same time the writer does, and so may be ignored by the proofs below. Readers setting their read flag during or after the first check but before the second check will be called *group 1* readers. Those setting their read flag during or after the second check but before the third check will be called *group 2* readers. *Group 3* readers are those setting their read flag during or after the third check by the writer.

Lemma 1: *Algorithm 1 preserves mutual exclusion between the writer and each reader on the backup buffers.*

> The proof of this lies in the use of a simple mutual exclusion protocol embedded in the algorithm. This protocol consists of a process signaling its interest in a resource, then checking if any other process has signaled interest

in the resource. By itself, this protocol leads to deadlock, but it does assure that mutual exclusion is maintained for the resource.

The only way in which a reader can possibly read the backup buffer is if it sees the write flag on. In phase 1, the phase in which the backup buffer is written, the writer has already checked for readers interested in the buffer pair. None were detected, so any reader that could possibly read the backup buffer cannot have finished setting its read flag before the writer tested it. Since the write flag is false when the writer begins the test, and the reader tests the write flag after it sets its read flag, the reader cannot see the write flag set until the writer sets it (that is, the reader cannot have seen an old value for the write flag without having been detected by the writer). Since the write flag is not set until after the backup buffer has been written, the reader will not attempt to read the backup buffer until the writer is through writing it. Therefore the backup buffer is never accessed by the writer and any reader at the same time.

Q.E.D.

Lemma 2: *Algorithm 1 preserves mutual exclusion between the writer and each reader on the primary buffers.*

By the writer's second check, all the phase 1 readers are gone, or else the writer would start fresh with a different buffer pair. Any phase 2 or phase 3 readers must see the write flag as true unless the writer has already finished writing the primary buffer. Thus the only way for any of these readers to access the primary copy is by seeing the forwarding bits set for some reader.

Phase 2 readers may see the forwarding bits set, set their forwarding bits and read the primary copy. If a reader has not finished by the writer's third check, the writer will see that reader's read flag set and abandon that pair of buffers. If any phase 2 reader that does finish before the third check sets its forwarding bits after the writer has cleared them, the writer will give up on that pair of buffers, so there is no possibility of any sort of chain of readers setting their forwarding bits after the writer has cleared them. In any case, when the writer successfully completes its third check of the read flags, all the phase two readers are finished and no phase three reader can see a pair of forwarding bits set that the writer will not also see (causing it to abandon that pair of buffers).

For the writer to continue, the forwarding bits must all be clear. Phase three readers will neither see the write flag off nor see any forwarding bits on until the writer has started to clear its write flag. Hence, none will set their forwarding bits until that time. Since this occurs after the writer has written the primary buffer, no reader can access the primary buffer while the writer is accessing it.

Q.E.D.

Lemma 3: *Algorithm 1 implements an atomic variable.*

What must be shown here is that if one read is strictly followed by another, the second read must obtain a value that is at least as new as the

first read.

There are two cases to consider: 1) the reads access the same buffer pair, and 2) the reads access different buffer pairs. In either case, the second reader must be "new" (meaning that it read the selector after the current write updated it), while the first reader may either be new or old (meaning that it read the selector after some earlier write to this pair of buffers).

Case 1: Same pair

The first read must get the newer value, so it must either read the primary buffer after the writer has written the new value to it and the second read must access the secondary buffer, or the first read must read the most recent previous value from the backup buffer and the second read obtain a very old value from the primary buffer. The second subcase implies that both the readers are old, so they must overlap, contradicting the premise that the second read is strictly after the first.

Therefore, consider the case in which the first reader reads the new value from the primary buffer. This means that the first reader must be a phase three reader, consequently, so must the second. Since the first read ends before the second read begins, the second reader must see the forwarding bits set by the first reader, so it will access the primary buffer. Note that this is the entire purpose of the forwarding bits: for the readers to be able to communicate about the value of the write flag that they saw, hence the copy that they read.

Case 2: Different pairs

For readers to access different buffer pairs, a write must be in progress sometime between the first read and the second read, possibly overlapping one or both of the reads. If the first reader gets the old pointer and the second reader gets the new pointer, then the oldest value that the second reader could return is that of the backup buffer, which is the same as the value in the primary buffer of the old pair. This in turn is the newest value that the first reader could return, so this subcase is not a problem.

If the first reader gets the new pointer and the second reader gets the old pointer from the selector, then they must both overlap the write of the selector. If this is the case, then the first reader must finish before the writer is through with this action, so it must see the write flag set. Any other reader that reads the write flag before the writer finishes writing the selector variable must also see the write flag set and will not change the forwarding bits: all of them must read the backup copy of the new buffer pair. Only a reader that reads the write flag strictly after the writer is through writing the selector variable will be able to see the write flag off and set its forwarding bits, but this is already too late to affect the behavior of any reader that completely finishes before the writer is through writing the selector variable. Thus the first reader must return the value from the backup buffer. The second reader (that obtains the old value for the selector) will either read the primary

copy of the old pair (which holds the same value as the backup copy of the new pair), or must overlap a later write to the the old pair. If the second read overlaps a later write to the old pair, then it may read the backup buffer of the old pair. However, the backup buffer will contain a value at least as recent as the primary buffer of the new pair because the later write will write a recent value in the backup before the reader can access it. In either case, the second reader must obtain a value at least as new as the primary buffer of the old pair, which holds the same value as the backup buffer of the new pair, so the second read returns a value at least as new as the first read.

Q.E.D.

Theorem 4: *Algorithm 1 using r+2 pairs of copies implements a wait-free, atomic, r-reader, 1-writer buffer using safe, r-reader, 1-writer bits.*

That the algorithm simulates an atomic variable has been shown by Lemma 3. That it is wait-free is evident if $r+2$ pairs of copies are used. The readers clearly never wait on the writer; they only decide which buffer of their chosen pair to read.

Once a write has started, any new reads that start will access the current buffer pair, which the writer avoids, until the writer is through writing the primary buffer of its chosen buffer pair. The writer can only be forced to abandon a buffer pair if it encounters a reader accessing it, and each reader can spoil at most one buffer pair in this manner. Since there are r readers, the writer can be forced to abandon at most r buffer pairs, and will avoid one more (the pair written by the preceding write). With $r+2$ buffer pairs, the writer must eventually find one pair free of readers by the pigeon-hole principle.

Q.E.D.

By varying the number of pairs of buffers used, this algorithm produces a spectrum of protocols that are wait-free for the readers, but provides a tradeoff for the writer between waiting and the number of buffers used. The tradeoff is identical to that obtained in [Newman-Wolfe '86a], mentioned earlier in this paper, except that the readers never wait.

Also worth mentioning is that the number of forwarding bits may be reduced if multi-writer, multi-reader regular bits are available. Instead of using a pair of distributed forwarding bits for each reader per buffer pair, only one of these more powerful forwarding bits for all the readers and a distributed bit for the writer be needed per pair of buffers. This does not reduce the order statistics for the distributed control bits needed, since each reader would still need an individual read bit per buffer pair.

Finally, it is possible for the writer to save its investment of time spent writing the backup buffer if the read flags are all clear at the third check. If only some forwarding bits have been set by phase 2 readers that have left before their read flags were checked, then the writer can clear the forwarding bits and attempt the third check again. This does not nullify Lemma 2, but it does make its proof slightly more complicated. Neither does it cause the writer to wait, although the writer may have to clear the forwarding bits several times before it succeeds. Since giving up entirely makes the proofs easier, this is the

only version shown in this paper.

Conclusions

An open problem of Lamport has been solved: a multi-reader, single-writer, atomic, multi-valued, wait-free register has been constructed using multi-reader, single-writer, safe bits. This improves on the atomic, multi-reader construction of Peterson in that it does not assume the use of multi-reader, atomic bits; it improves on the construction of Lamport in that it provides a multi-reader register instead of a single- reader register.

The question of how to construct multi-writer, multi-reader, atomic, wait-free registers from weaker registers remains open. It appears that the techniques used for the construction presented here are not directly extensible to the multi-writer case.

In submissions to this conference, Peterson and Burns claim solutions to both the single-writer and the multi-writer problem [Peterson & Burns '86], [Peterson & Burns '87]. Their solution to the single-writer problem uses regular, multi-reader, multi-valued registers. When directly reduced to safe, multi-reader bits using the most favorable techniques (either those of [Lamport '85] or an adaptation of [Peterson '83a], depending on the number of readers and the number of values the variable must hold), their solution uses $2(b + 2)(r + 2) + 6r - 2$ safe bits, while the solution presented here uses $(r + 2)(3r + 2 + 2b) - 1$ safe bits. Their new solution may be used to simulate the atomic bit required by [Peterson '83a] to produce a solution that only uses $(r + 2)b + 10r + 5$ safe, multi-reader bits. Thus, the solution of [Peterson & Burns '87] is more space-efficient than the one presented here. The only objection to their solution is that the writer is forced to make a separate copy for each reader that starts after the writer last made that reader a private copy. This means that the writer may have to make copies for readers that are not active at the same time as the writer. The solution presented here may also require the writer to make as many as $r+1$ copies of the buffer, but all of these above the minimum of two are caused by active readers. This raises the interesting question of a tradeoff between space used and the amount of work required of the readers and the writer, a topic for future study.

Acknowledgements

I would like to thank G. L. Peterson and J. E. Burns for pointing out bugs in an earlier version of the protocol. The error found by Burns was both subtle and interesting in that it required two variables to be 'flickering' at the same time for the incorrect results to occur. The incorrect protocol thus earned the dubious distinction of being the first protocol of which he or I am aware that requires two variables to flickering simultaneously in order to fail. A. Deshpande has also participated in helpful discussions on the topic of synchronization.

References

[1] Burns, J. E. and G. L. Peterson, ['87], "Constructing Multi-reader Atomic Values From Non-atomic Values," *Proceedings of PODC 1987.*

[2] Courtois, P. J., F. Heymans and D. L. Parnas['71], "Concurrent Control with 'Readers' and 'Writers'," *CACM* 14:10, 667-668, 1971.

[3] Dijkstra, E. W.['65], "Solution of a Problem in Concurrent Programming Control," *CACM* 8:9, 569, 1965.

[4] Lamport, L.['74], "A New Solution of Dijkstra's Concurrent Programming Problem," *CACM* 17:8, 453-455, 1974.

[5] Lamport, L.['77], "Concurrent Reading and Writing," *CACM* 20:11, 806-811, 1977.

[6] Lamport, L.['85], "On Interprocess Communication," DEC Systems Research Center, SRC Report 8, Dec. 25, 1985.

[7] Newman-Wolfe, R. E. ['86a], "Economical Atomic Multi-Reader Shared Registers," *Proceedings of the 24th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL, October 1986.

[8] Newman-Wolfe, R. E. ['86b], "Communication Issues in Parallel Computation," University of Rochester Computer Science Department, TR-200, December, 1986.

[9] Peterson, G. L.['83a], "Concurrent Reading While Writing," *TOPLAS* 5:1, 46-55, 1983.

[10] Peterson, G. L.['83b], "A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables," *TOPLAS* 5:1, 56-65, 1983.

[11] Peterson, G. L. and J. E. Burns ['86], "Concurrent Reading While Writing: the Multi-writer Case," TR GIT-ICS 86/26, School of Information and Computer Science, Georgia Institute of Technology, Dec. 1986.

[12] Vitanyi, P. and B. Awerbuch ['86], "Atomic Shared Register Access by Asynchronous Hardware," *Proceedings of the 27th FOCS*, 233-243, 1986.

Writer:
Get index of current buffer pair.
Find a buffer pair with no readers signaling interest,
other than the current pair.
Write the most recent previous value to the backup buffer.
Signal interest in this pair of buffers.
If there are now readers showing interest in this pair,
then remove notice of interest and start over. Otherwise,
Clear all the forwarding bit pairs for this buffer pair.
If there are any readers showing interest in this pair of buffers
or if any forwarding bits are set
then remove notice of interest and start over. Otherwise,
Write the new value to the primary buffer.
Change the index.
Remove interest notice.

Reader j:
Get index of current buffer pair.
Signal interest in this pair.
If writer does not show interest in this pair
or if any other reader's forwarding bits are set, then
set the jth forwarding bits for this buffer pair.
If the jth forwarding bits for this pair are set,
then **read the primary copy.**
else **read the backup copy.**
Remove notice of interest.

Fig.1. High-level view of the algorithm

(* **Constants** *)

NR : integer; (* *the number of readers**)
M : integer; (* *the number of pairs of buffers**)

(* **Shared variables** *)

BN : regular, distributed, M-valued register; (**the selector**)
R[M][NR] : regular, distributed bits; (**reader i's read flag for buffer j is R[j][i]**)
W[M] : regular, distributed bits; (**write flag for buffer j is W[j]**)
FR[M][NR] : regular, distributed bits; (**readers' forwarding bits**)
FW[M][NR] : regular, distributed bits; (**writer's forwarding bits**)
Primary[M], Backup[M] : safe, distributed bits; (**the buffer pairs**)

Fig.2. Shared variables and constants for low-level view of algorithm. All other variables used (which start with a lower case letter) are local variables. All variables are initialized to 0 or False.

```
PROC Write(newval)
 BEGIN
 newbuf := prev := BN;
 gotOne := False;
 WHILE (!gotOne) DO
     BEGIN
     newbuf := FindFree(prev, newbuf);     (* first check *)
     gotOne := True;
     Backup[newbuf] := oldval;
     W[newbuf] := True;
     IF (! Free(newbuf)) THEN          (* second check *)
       BEGIN
       W[newbuf] := False;
       gotOne := False;
       continue; (* go back to top of loop *)
       END;
     ClearForwards(newbuf);
     IF (! Free(newbuf)) THEN          (* third check *)
       BEGIN
       W[newbuf] := False;
       gotOne := False;
       continue; (* go back to top of loop *)
       END;
     IF (ForwardSet(newbuf)) THEN
       BEGIN
       W[newbuf] := False;
       gotOne := False;
       continue; (* go back to top of loop *)
       END;
     END; (* do *)
 Primary[newbuf] := newval;
 BN := newbuf;
 W[newbuf] := False;
 oldval := newval;
 END;
```

Fig.3. Low level view of the writer's protocol. Oldval is assumed to have been initialized by the previous write.

```
PROC ClearForwards(bufno)
 BEGIN
 FOR (i:=1 to NR) DO
     FW[bufno][i] := FR[bufno][i];
 END;


BOOL Free(bufno)
 BEGIN
 FOR (i:=1 to NR) DO
     IF (R[bufno][i]) THEN
       RETURN(False);
 RETURN(True);
 END;


INT FindFree(current, bufno)
 BEGIN
 j := bufno;
 WHILE (True) DO
     BEGIN
     IF ( (j != current) AND Free(j) ) THEN
       RETURN(j);
     ELSE
       j := j+1 MOD M;
     END;
 END;
```

Fig.4. Procedures used by the writer

```
BUF Read(i)            (* reader number i *)
 BEGIN
 current := BN;
 R[current][i] := True;
 IF ((W[current] === False) OR (ForwardSet(current))) THEN
     BEGIN
     FR[current][i] := !FW[current][i];
     value := Primary[current];
     END;
 ELSE     value := Backup[current];
 R[current][i] := False;
 RETURN(value);
 END;


BOOL ForwardSet(bufno)
 BEGIN
 FOR (i:=1 to NR) DO
     IF (FR[bufno][i] != FW[bufno][i]) THEN
       RETURN(True);
 RETURN(False);
 END;
```

Fig.5. Low level view of the readers' protocol and procedures